Probabilistic Programming Languages

Guillaume Baudart

MPRI 2024-2025

Warm-up: Rejection sampling

Rejection sampling (hard)


```
module Rejection_sampling_hard : sig
  val sample : 'a Distribution.t → 'a
  val assume : bool → unit
  val infer : ?n:int → ('a → 'b) → 'a → 'b Distribution.t
end = struct ... end
```

Inference algorithm

- Run the model to get a sample
- `sample` : draw a value from a distribution
- `assume` : accept / reject a sample
- If a sample is rejected, re-run the model to get another sample

Hard conditioning

- `val observe : 'a Distribution.t → 'a → unit`
- Assume that a value was sampled from a distribution (??)

Importance sampling

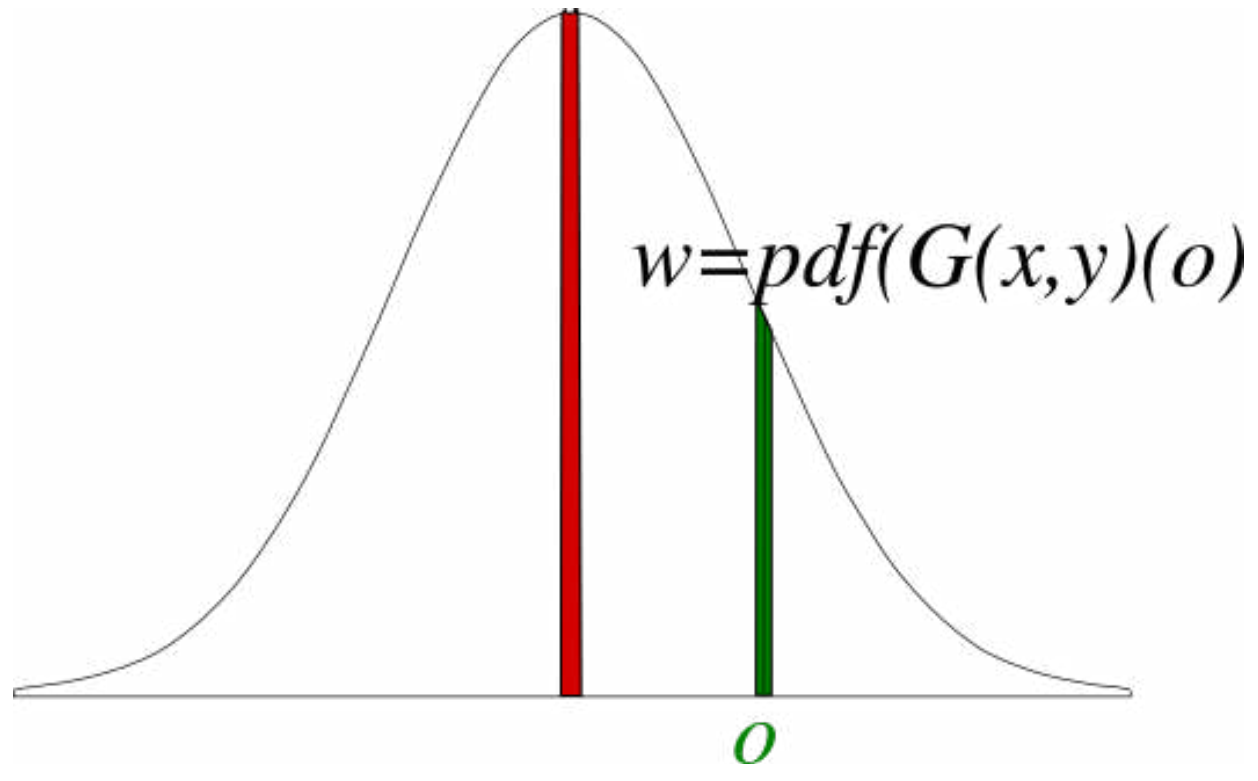

```
module Importance_sampling : sig
  type prob
  val sample : prob → 'a Distribution.t → 'a
  val factor : prob → float → unit
  val infer : ?n:int → (prob → 'a → 'b) → 'a → 'b Distribution.t
end = struct ... end
```

Inference algorithm

- Run a set of n independent executions
- `sample`: draw a sample from a distribution
- `factor`: associate a score to the current execution
- Gather output values and score to approximate the posterior distribution

Likelihood weighting

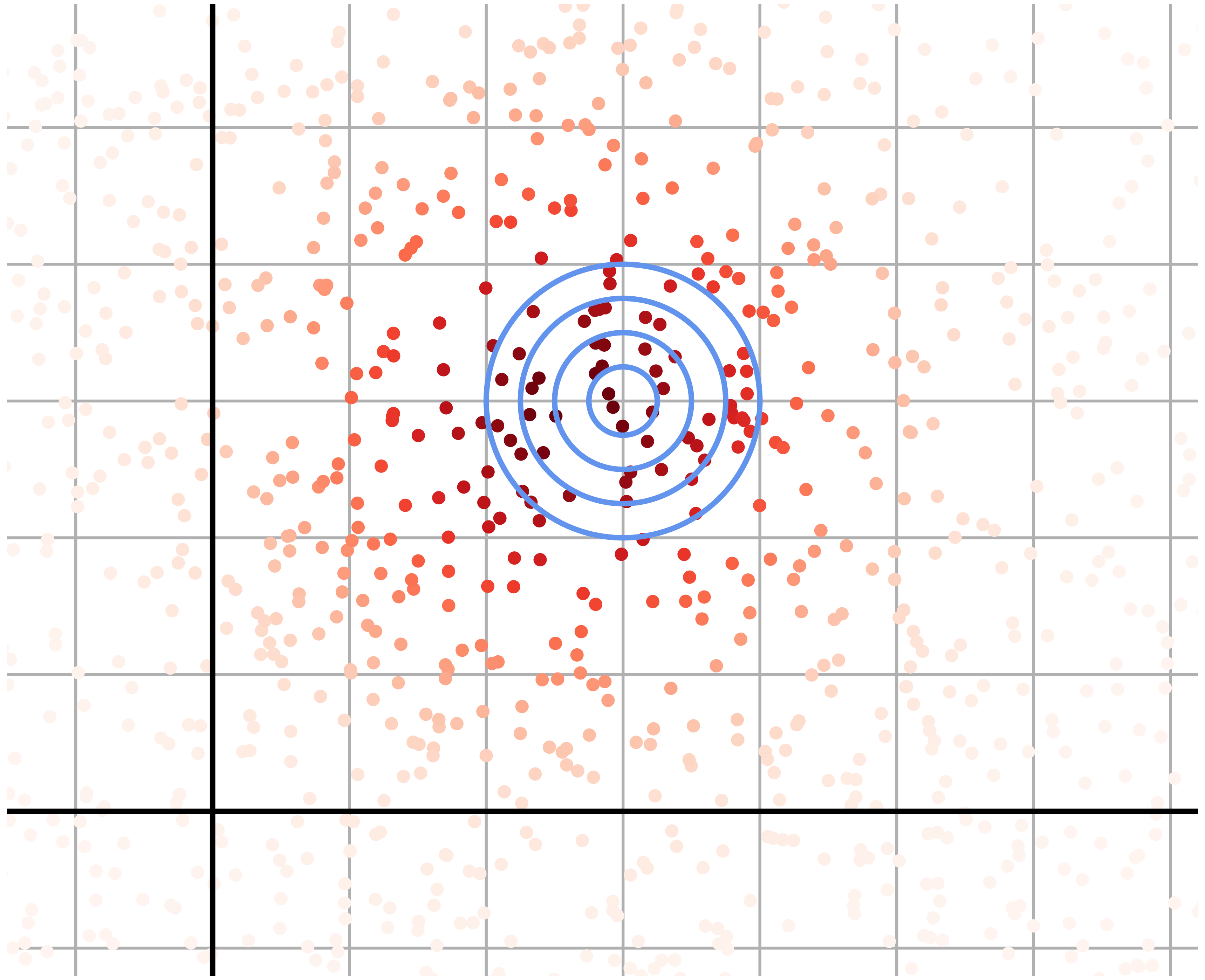
- `observe d x := factor (logpdf d x)`



Example: Noisy position

```
open Basic.Importance_sampling

let gauss obs =
  let x = sample (gaussian ~mu:0.0 ~sigma:10.0) in
  let y = sample (gaussian ~mu:0.0 ~sigma:10.0) in
  List.iter
    (fun (xo, yo) →
      observe (gaussian ~mu:x ~sigma:1.0) xo;
      observe (gaussian ~mu:y ~sigma:1.0) yo )
    obs;
  (x, y)

let _ =
  let dist = infer gauss data in
  plot dist
```

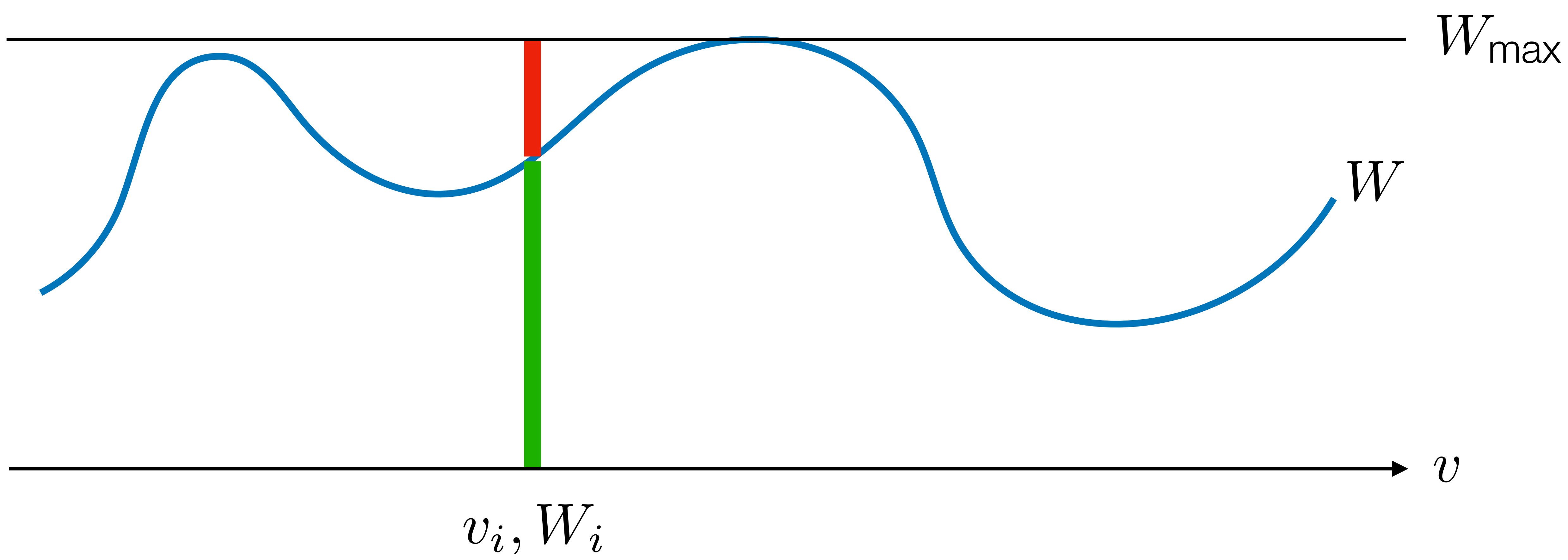
10 000 particles



Weighted rejection sampling

Adapt rejection sampling to soft conditioning

- Execute the sampler to get a pair (v_i, W_i)
- Suppose W_{max} is known
- Accept the sample with probability W_i / W_{max} or retry



But $W_{\max}$ is not known...

Try it in BYO-PPL!

Rejection sampling

basic.ml

```
module Rejection_sampling = struct
  include Importance_sampling

  let infer ?(n = 1000) ?(max_score = 0.) model data =
    let rec gen i =
      let prob = { score = 0. } in                                  (* reset the score *)
      let value = model prob data in                                  (* run the model *)
      let alpha = exp (min 0. (prob.score -. max_score)) in
      let u = Random.float 1. in
      if u ⩽ alpha then value else gen i                            (* accept / reject *)
    in
    let samples = List.init n gen in
    Distribution.empirical ~samples
end
```

The curse of dimensionality

Problem becomes harder as the dimension increases

Basic inference: importance sampling

- Performances decrease exponentially when the dimension increases
- Only use for low-dimension models

How to mitigate this problem?

- Make assumptions about the posterior distributions
- Break the problem into simpler, smaller problems

17h45mn

Sven Sachsalber, *Looking for a needle in the Haystack*, 2014

Markov Chain Monte Carlo

Probabilistic Programming Languages

Markov Chain Monte Carlo (MCMC)

Main idea

- Create a Markov chain that converge to the posterior distribution
- Iterate the process until convergence
- Generate samples to approximate the distribution

Pros

- Faster convergence
- Better results for high-dimensional models
- Advanced state-of-the-art optimizations (e.g., HMC, NUTS).

Cons

- Convergences?
- Traps: multimodal, funnel
- Samples correlation

Metropolis Hastings

Trace

- X: set of random variables (`sample`)
- $P(X)$: prior distribution of X
- A trace characterize one posible execution
- W: score of the execution (same as importance sampling)

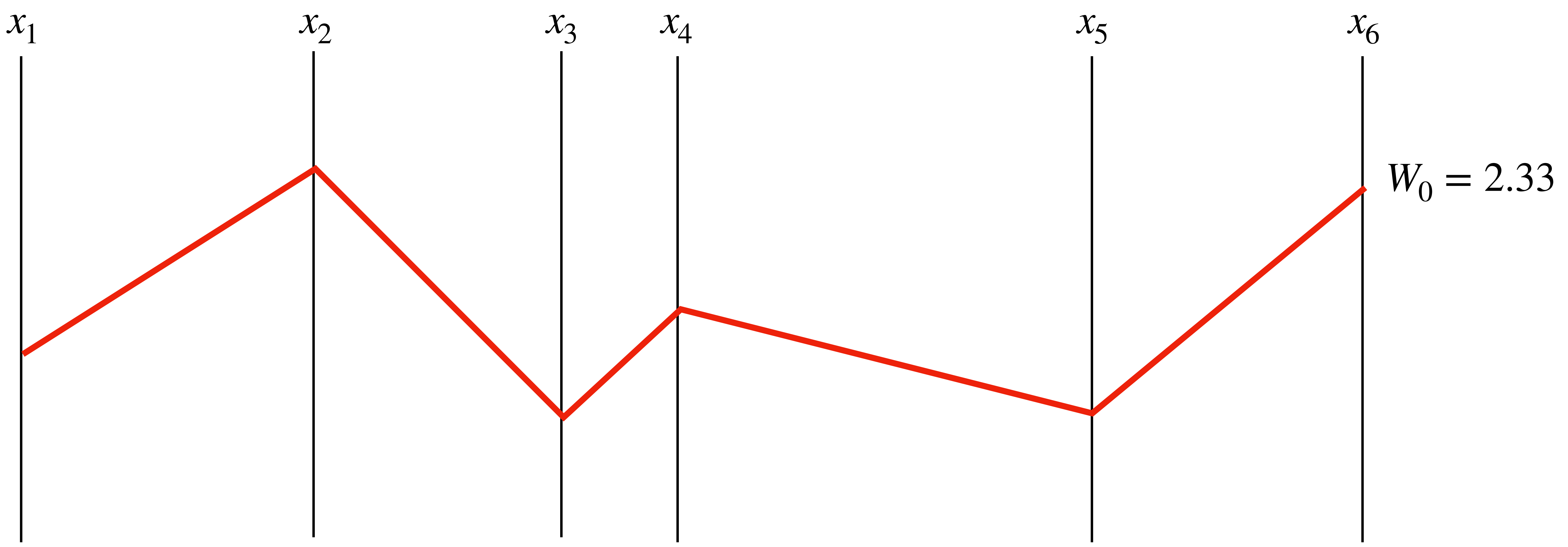
Metropolis-Hastings algorithm

- Initialization: draw X_0 at random to get a pair (v_0, W_0).
- At each step:
  1. Draw a *candidate* $X' \sim Q(X' \mid X_i)$ to get (v', W')
  2. Acceptance rate: $\alpha = \dfrac{P(X')\, W'\, Q(X_i \mid X')}{P(X_i)\, W_i\, Q(X' \mid X_i)}$.
  3. Draw $u \sim U(0, 1)$.
  4. If $u \leq \alpha$ (*accept*) $\begin{cases} X_{i+1} & = & X' \\ v_{i+1} & = & v' \\ W_{i+1} & = & W' \end{cases}$ else (*reject*) $\begin{cases} X_{i+1} & = & X_i \\ v_{i+1} & = & v_i \\ W_{i+1} & = & W_i \end{cases}$

Multi-sites Metropolis Hastings

Markov chain on execution traces

- Execute the sampler to get a candidate X' and the associated value and score (v_i, W_i)
- If $W' \geq W_i$ accept the candidate (and the associated value)
- Else accept the candidate with probability W'/W_i
- Otherwise keep the previous trace X_i



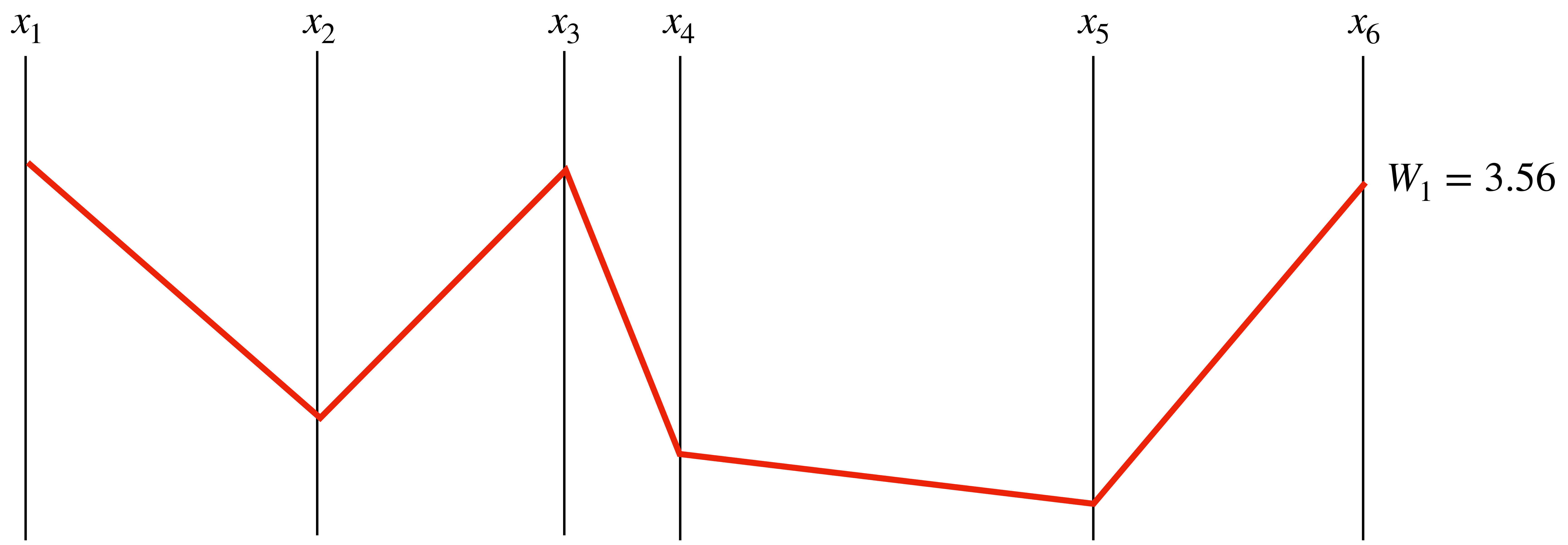
outputs

v_0

Multi-sites Metropolis Hastings

Markov chain on execution traces

- Execute the sampler to get a candidate X' and the associated value and score (v_i, W_i)
- If $W' \geq W_i$ accept the candidate (and the associated value)
- Else accept the candidate with probability W'/W_i
- Otherwise keep the previous trace X_i



outputs

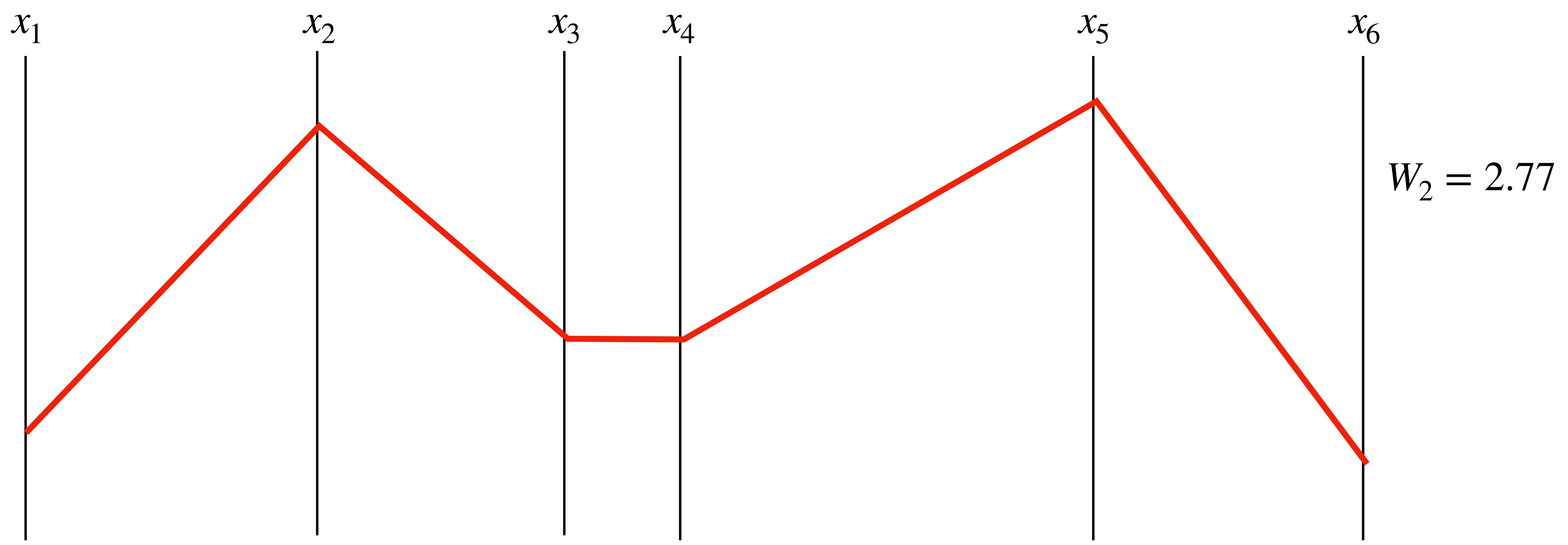
v_0

v_1

Multi-sites Metropolis Hastings

Markov chain on execution traces

- Execute the sampler to get a candidate X' and the associated value and score (v_i, W_i)
- If $W' \geq W_i$ accept the candidate (and the associated value)
- Else accept the candidate with probability W'/W_i
- Otherwise keep the previous trace X_i



outputs

v_0

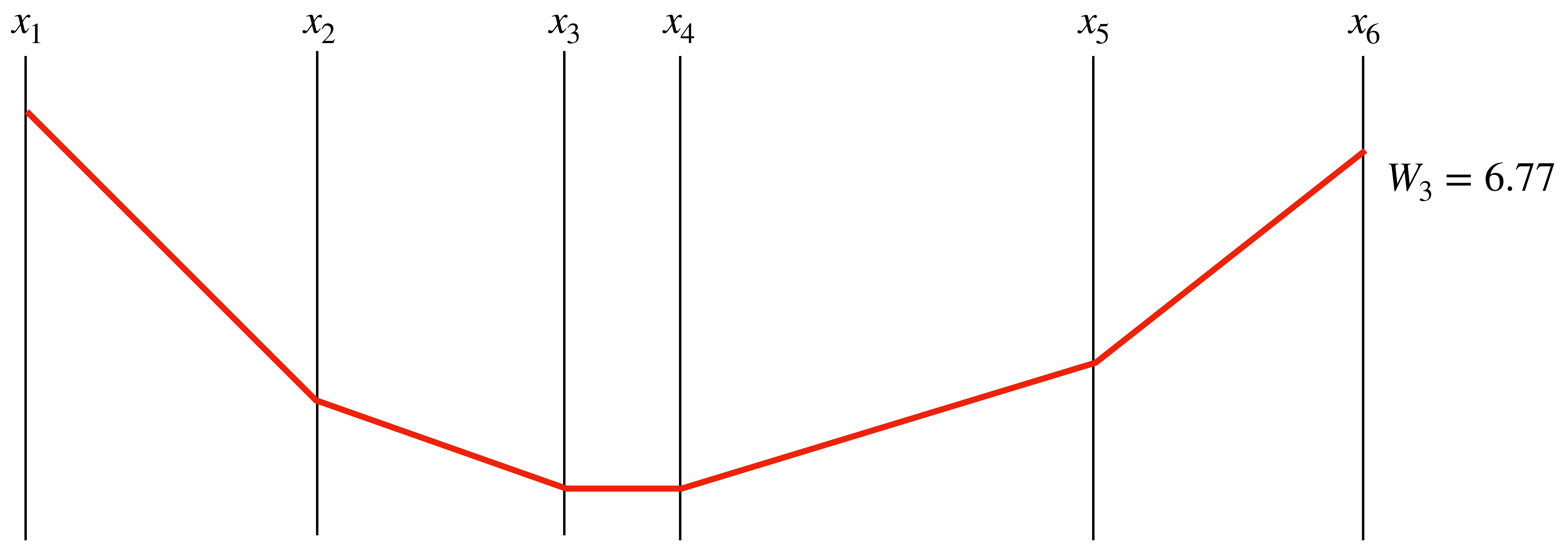
v_1

v_1

Multi-sites Metropolis Hastings

Markov chain on execution traces

- Execute the sampler to get a candidate X' and the associated value and score (v_i, W_i)
- If $W' \geq W_i$ accept the candidate (and the associated value)
- Else accept the candidate with probability W'/W_i
- Otherwise keep the previous trace X_i



outputs

v_0

v_1

v_1

v_3

...

Multi-sites Metropolis Hastings: acceptation

Re-execute the entire trace

- Draw proposal from priors $Q(X' \mid X_i) = P(X_i)$
- Resample all variables in X_i at each iteration

$$\begin{aligned} \alpha &= \frac{P(X')\, W'\, Q(X_i \mid X')}{P(X_i)\, W_i\, Q(X' \mid X_i)} \\ &= \frac{P(X')\, W'\, P(X_i)}{P(X_i)\, W_i\, P(X')} \\ &= \frac{W'}{W_i} \end{aligned}$$

Markov chain on execution traces

- Execute the sampler to get a candidate X' and the associated value and score (v_i, W_i)
- If $W' \geq W_i$ accept the candidate (and the associated value)
- Else accept the candidate with probability W'/W_i
- Otherwise keep the previous trace X_i

Try it in BYO-PPL!

Multi-sites Metropolis Hastings

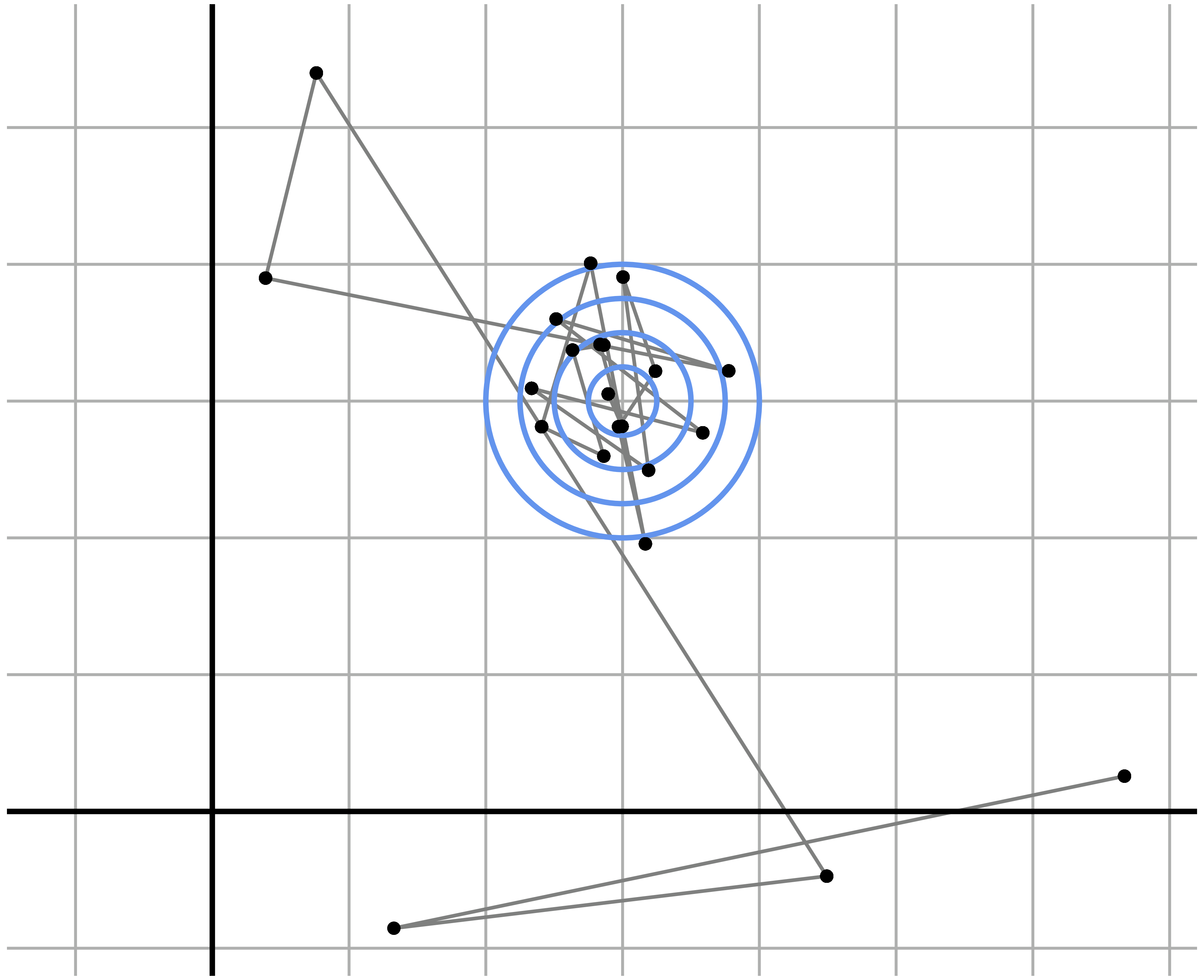
basic.ml

```
module Simple_metropolis = struct
  include Importance_sampling

  let infer ?(n = 1000) model data =
    let rec gen n samples old_score old_value =
      if n = 0 then samples                                (* return the list of samples *)
      else
        let prob = { score = 0. } in                                        (* reset prob *)
        let new_value = model prob data in                          (* generate candidate *)
        let alpha = exp (min 0. (prob.score -. old_score)) in
        let u = Random.float 1.0 in
        if not (u < alpha) then
          gen (n - 1) (old_value :: samples) old_score old_value                (* reject *)
        else gen (n - 1) (new_value :: samples) prob.score new_value            (* accept *)
    in
    let prob = { score = 0. } in
    let first_value = model prob data in                          (* generate first trace *)
    let samples = gen n [] prob.score first_value in                  (* generate samples *)
    Distribution.empirical ~samples
end
```

Example: Noisy position

```
open Basic.Simple_metropolis

let gauss obs =
  let x = sample (gaussian ~mu:0.0 ~sigma:10.0) in
  let y = sample (gaussian ~mu:0.0 ~sigma:10.0) in
  List.iter
    (fun (xo, yo) →
      observe (gaussian ~mu:x ~sigma:1.0) xo;
      observe (gaussian ~mu:y ~sigma:1.0) yo )
    obs;
  (x, y)

let _ =
  let dist = infer gauss data in
  plot dist
```

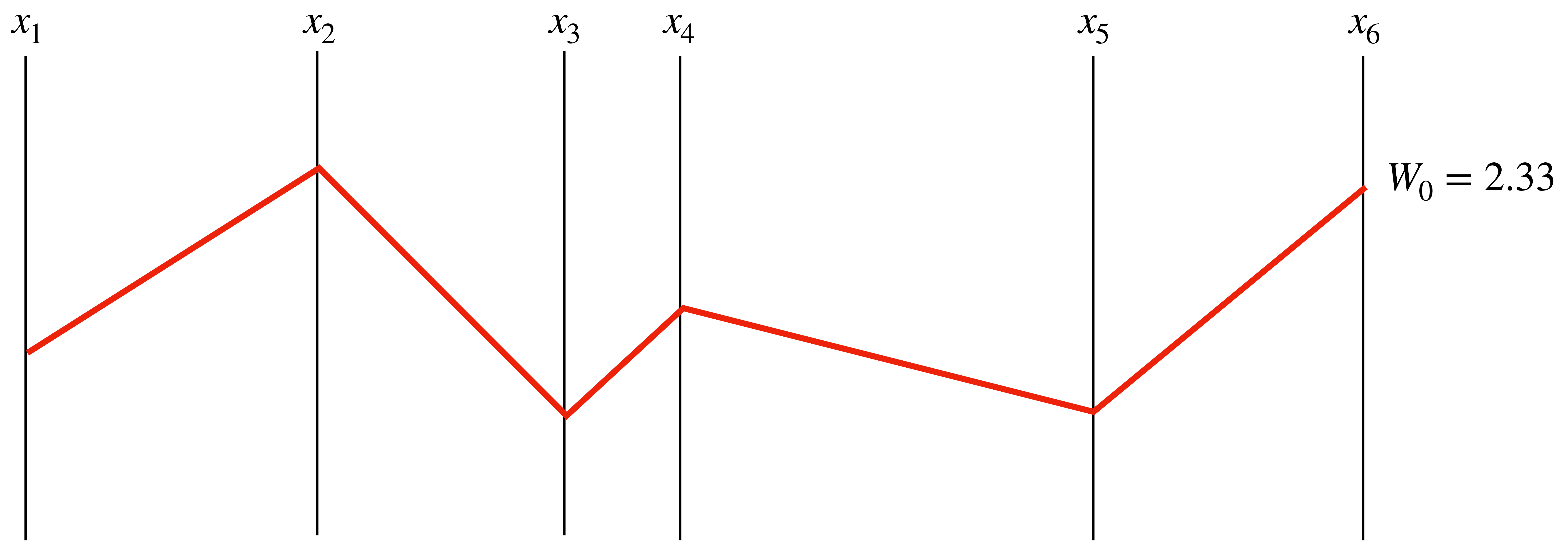
7000 samples



Single-site Metropolis Hastings

Reuse most of the previous trace (i.e., sampled values)

- Choose one random variable x_{regen} to resample to obtain a new execution
- Accept the trace with probability α
- Otherwise use the previous trace



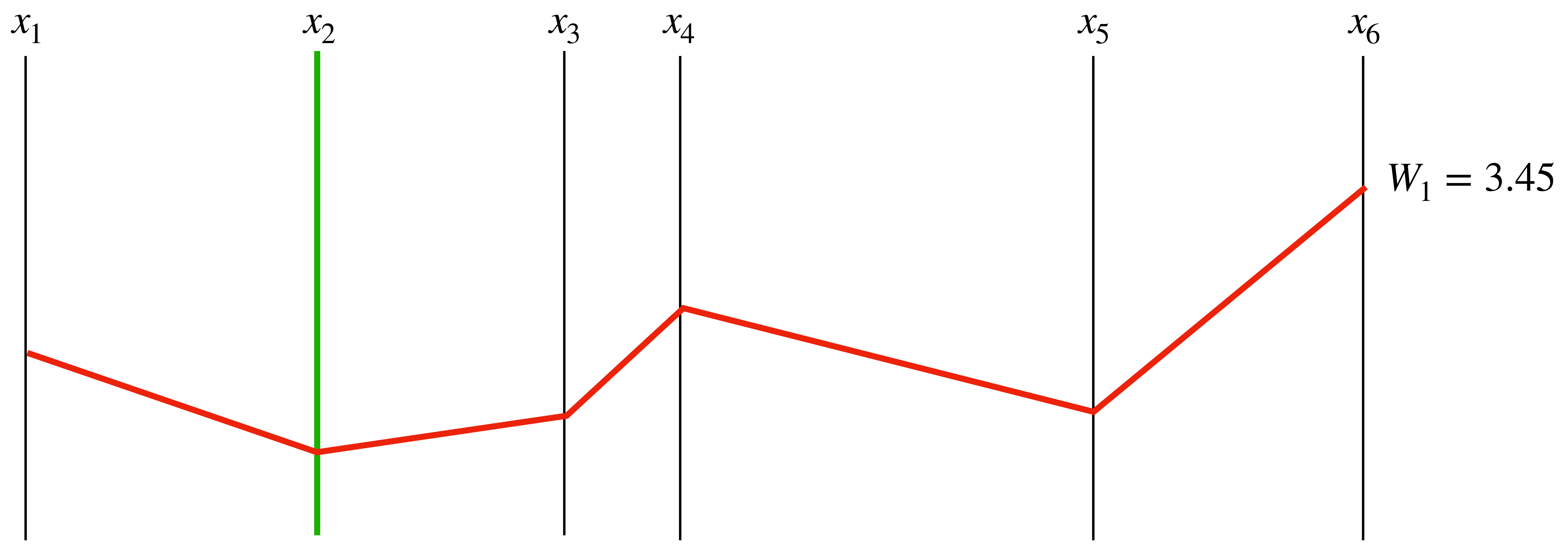
samples

v_0

Single-site Metropolis Hastings

Reuse most of the previous trace (i.e., sampled values)

- Choose one random variable x_{regen} to resample to obtain a new execution
- Accept the trace with probability α
- Otherwise use the previous trace



samples

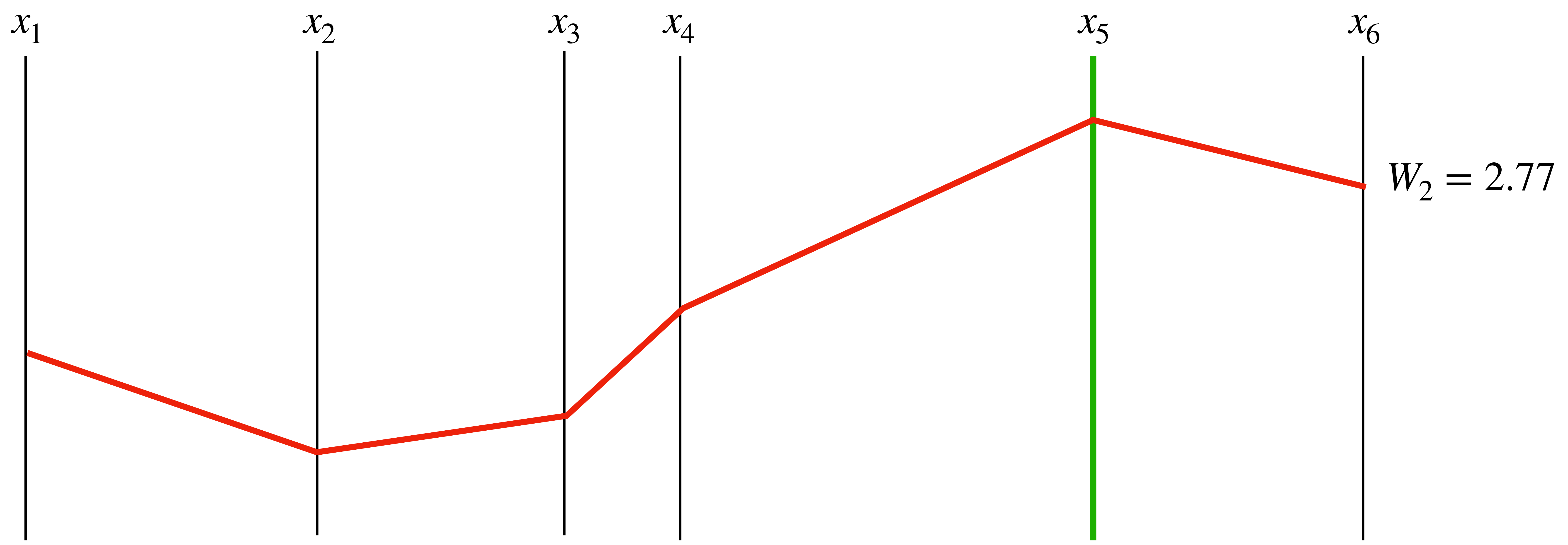
v_0

v_1

Single-site Metropolis Hastings

Reuse most of the previous trace (i.e., sampled values)

- Choose one random variable x_{regen} to resample to obtain a new execution
- Accept the trace with probability α
- Otherwise use the previous trace



samples

v_0

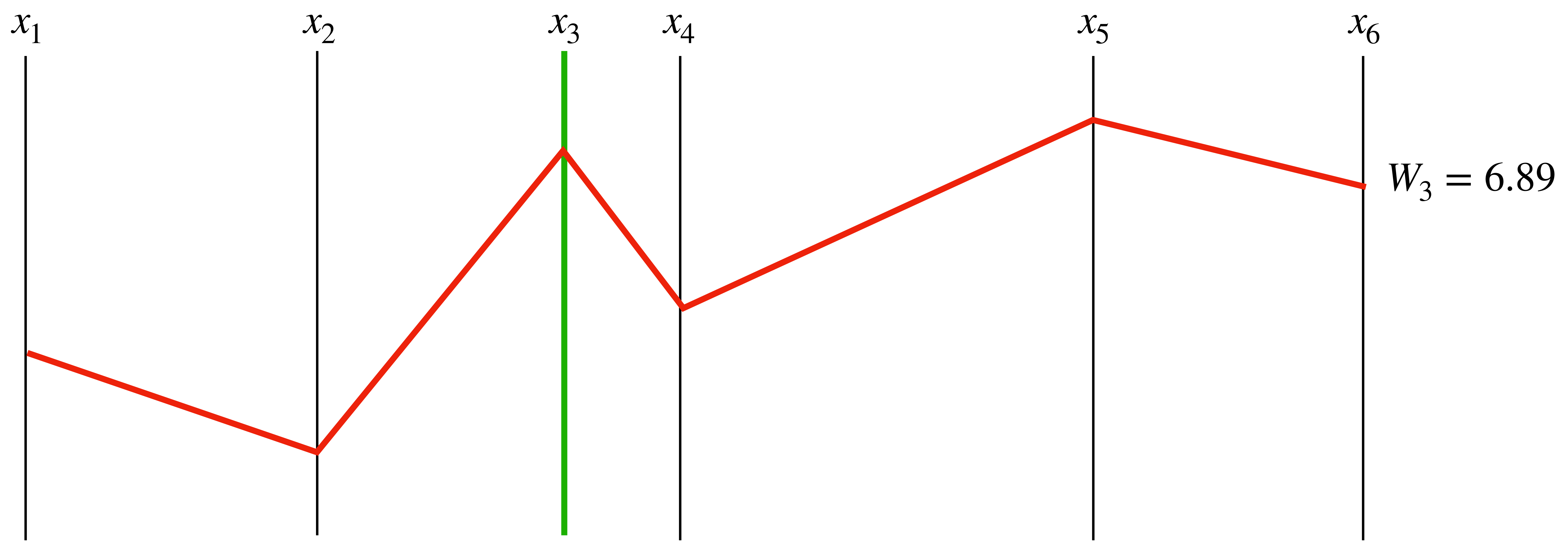
v_1

v_1

Single-site Metropolis Hastings

Reuse most of the previous trace (i.e., sampled values)

- Choose one random variable x_{regen} to resample to obtain a new execution
- Accept the trace with probability α
- Otherwise use the previous trace



samples

v_0

v_1

v_1

v_3

Single-site Metropolis Hastings: acceptation

Notations

- For $x \in X$, $w(x)$: density of sample x (same as `observe`)
- $C = (X' \cap X - \{x_{\text{regen}}\})$: *cache*, i.e., reused variables between X and X'

$$P(X) = \prod_{x \in X} w(x)$$

prior distribution

$$Q(X' \mid X) = \frac{1}{|X|} \prod_{x \in (X' - C)} w'(x)$$

choice of X' from X

$$\begin{aligned}
\alpha &= \frac{P(X')\, W'\, Q(X_i \mid X')}{P(X_i)\, W_i\, Q(X' \mid X_i)} \\
&= \frac{\prod_{x \in X'} w'(x)}{\prod_{x \in X_i} w(x)} \frac{W'}{W_i} \frac{|X_i| \prod_{x \in (X_i - C)} w(x)}{|X'| \prod_{x \in (X' - C)} w'(x)} \\
&= \frac{|X_i|}{|X'|} \frac{W'}{W_i} \frac{\prod_{x \in C} w'(x)}{\prod_{x \in C} w(x)}
\end{aligned}$$

Reused variables are treated as observations

Single-site Metropolis Hastings

Rerun (part of) the trace

- Assign a unique name to each random variable sample
- Can be added by a compiler: addressing transform
- Store sample and score in a table (cache)

```
let gauss obs =
  let x = sample (gaussian ~mu:0.0 ~sigma:10.0) "x" in
  let y = sample (gaussian ~mu:0.0 ~sigma:10.0) "y" in
  List.iter
    (fun (xo, yo) →
      observe (gaussian ~mu:x ~sigma:1.0) xo;
      observe (gaussian ~mu:y ~sigma:1.0) yo )
    obs;
  (x, y)
```

Single-site Metropolis Hastings


```
module Metropolis_hastings = struct
  type 'a sample_site = { x_value : 'a; x_score : float }
  type 'a prob = {
    mutable score : float;                                        (* current score *)
    x_store : (string, 'a sample_site) Hashtbl.t;            (* sample store (trace) *)
    cache : (string, 'a sample_site) Hashtbl.t;                             (* cache *)
  }

  let sample prob d name =
    let x_value =
      match Hashtbl.find_opt prob.cache name with
      | Some { x_value; _ } → x_value                           (* reuse if possible *)
      | None → Distribution.draw d                        (* otherwise draw a sample *)
    in
    let x_score = Distribution.logpdf d x_value in
    Hashtbl.add prob.x_store name { x_value; x_score };       (* store sample site *)
    x_value
```

Single-site Metropolis Hastings


```
let mh cache old_score old_x_store score x_store =
  let l_alpha = log (length old_x_store)) -. log (length x_store)) in
  let l_alpha = l_alpha +. score -. old_score in
  let dom = intersect cache x_store in
  let l_alpha = List.fold_left
    (fun l_alpha x →
      let { x_score; _ } = Hashtbl.find x_store x in
      let { x_score = old_x_score; _ } = Hashtbl.find old_x_store x in
      l_alpha +. x_score -. old_x_score)
    dom l_alpha
  in
  exp (min 0. l_alpha)
```

$$\alpha = \frac{|X_i|}{|X'|} \frac{W'}{W_i} \frac{\prod_{x \in C} w'(x)}{\prod_{x \in C} w(x)}$$

Single-site Metropolis Hastings


```
  let infer ?(n = 1000) model data =
    let rec gen n samples old_score old_value old_x_store =
      [ ... ]
    in

    let prob = { score = 0.; x_store = empty; cache = empty} in
    let first_value = model prob data in                          (* generate first trace *)
    let samples = gen n [] prob.score first_value prob.x_store in     (* generate samples *)
    Distribution.empirical ~samples
end
```

Same as multi-sites...

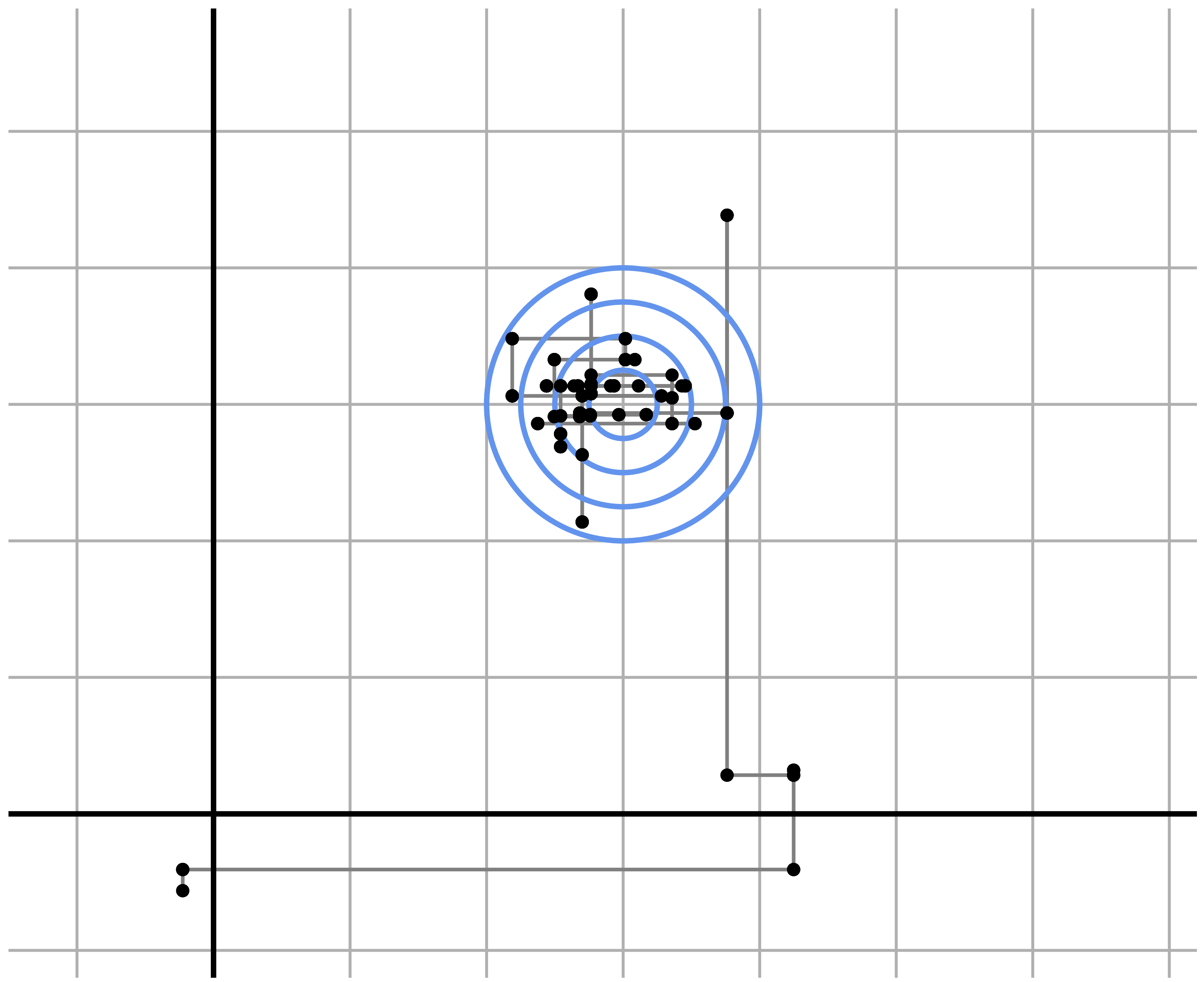
Single-site Metropolis Hastings


```
let rec gen n samples old_score old_value old_x_store =
  if n = 0 then samples                             (* return the list of samples *)
  else
    let regen = pick old_x_store in                              (* pick one sample *)
    let cache = Hashtbl.copy old_x_store in          (* use previous store as cache *)
    Hashtbl.remove cache regen;                       (* force resampling for regen *)

    let prob = { score = 0.; x_store = empty; cache } in            (* reset prob *)
    let new_value = model prob data in                       (* generate candidate *)
    let alpha = mh cache old_score old_x_store prob.score prob.x_store in
    let u = Random.float 1.0 in
    if not (u < alpha) then
      gen (n - 1) (old_value :: samples) old_score old_value old_x_store     (* reject *)
    else gen (n - 1) (new_value :: samples) prob.score new_value prob.x_store (* accept *)
in
```

Example: Noisy position

```
open Basic.Metropolis_hastings

let gauss obs =
  let x = sample (gaussian ~mu:0.0 ~sigma:10.0) in
  let y = sample (gaussian ~mu:0.0 ~sigma:10.0) in
  List.iter
    (fun (xo, yo) →
      observe (gaussian ~mu:x ~sigma:1.0) xo;
      observe (gaussian ~mu:y ~sigma:1.0) yo )
    obs;
  (x, y)

let _ =
  let dist = infer gauss data in
  plot dist
```

1000 samples



Limitations

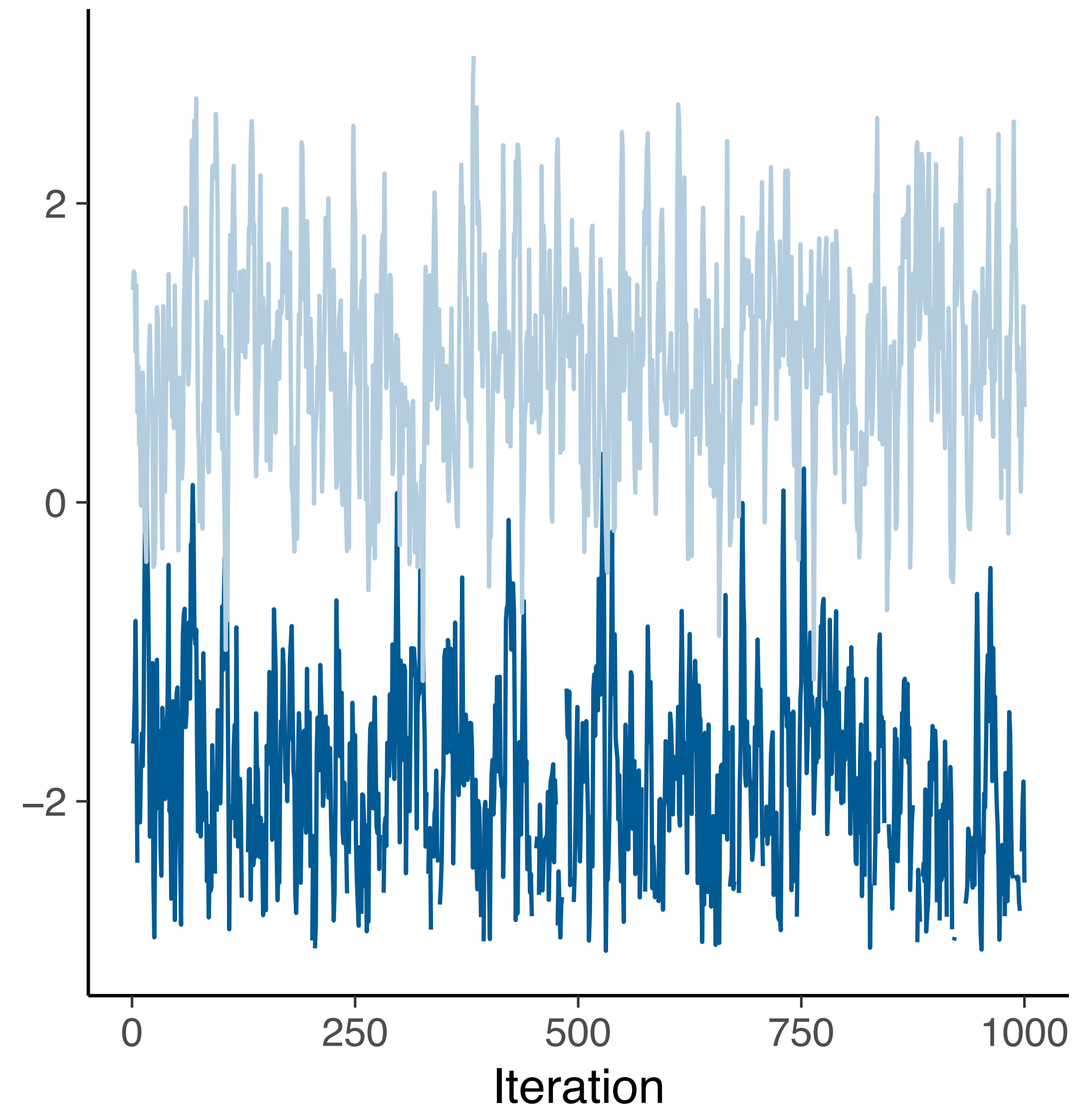
Convergence: theoretical conditions are complex

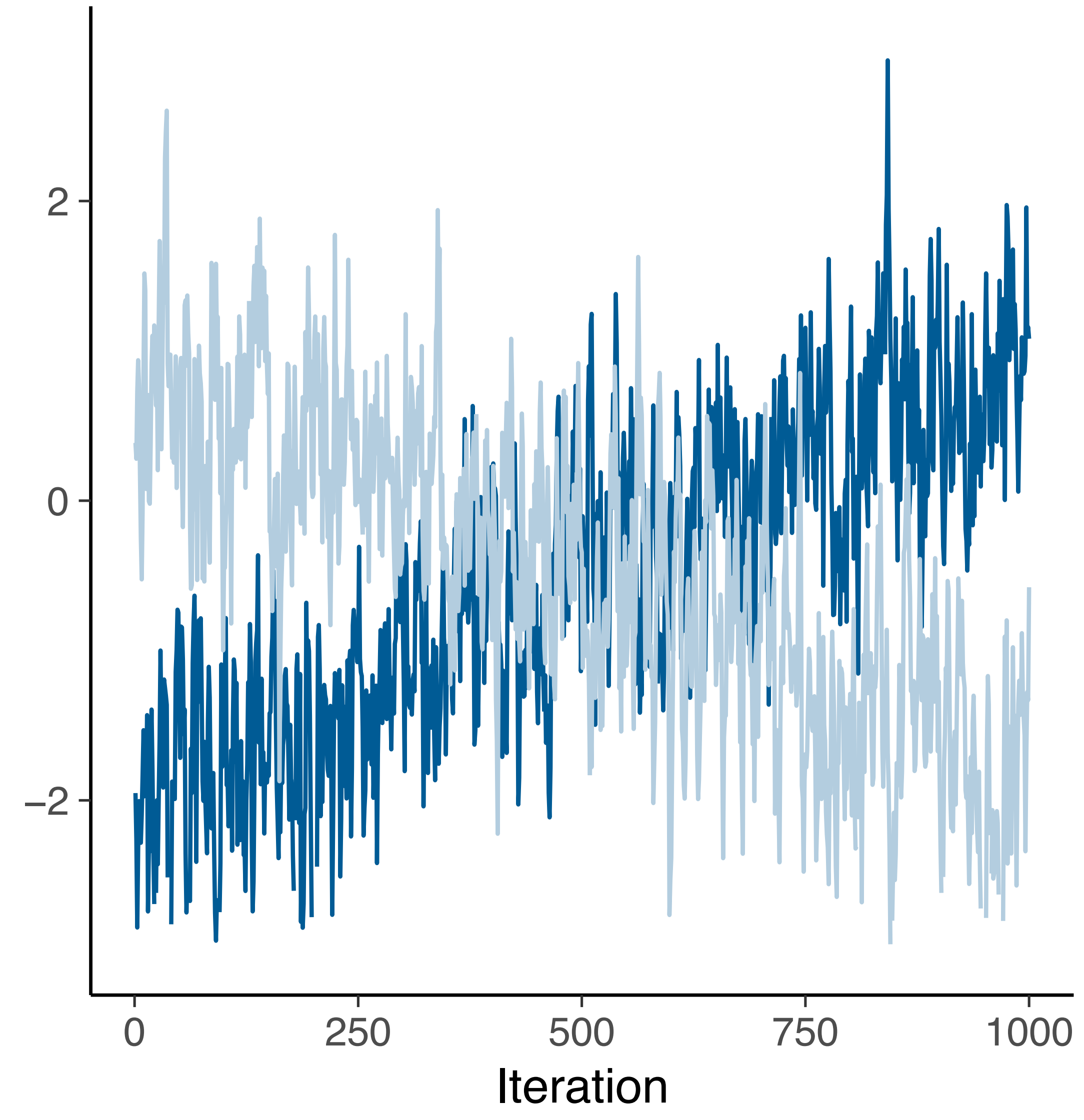
- Check experimentally: trace plot, R-hat (multi-chains)
- Solution: warmup, change initial conditions, reparameterization, ...

Sample correlation

- Diagnostic tools ESS (effective sample size)
- Solution: thinning, (keep one sample every n)

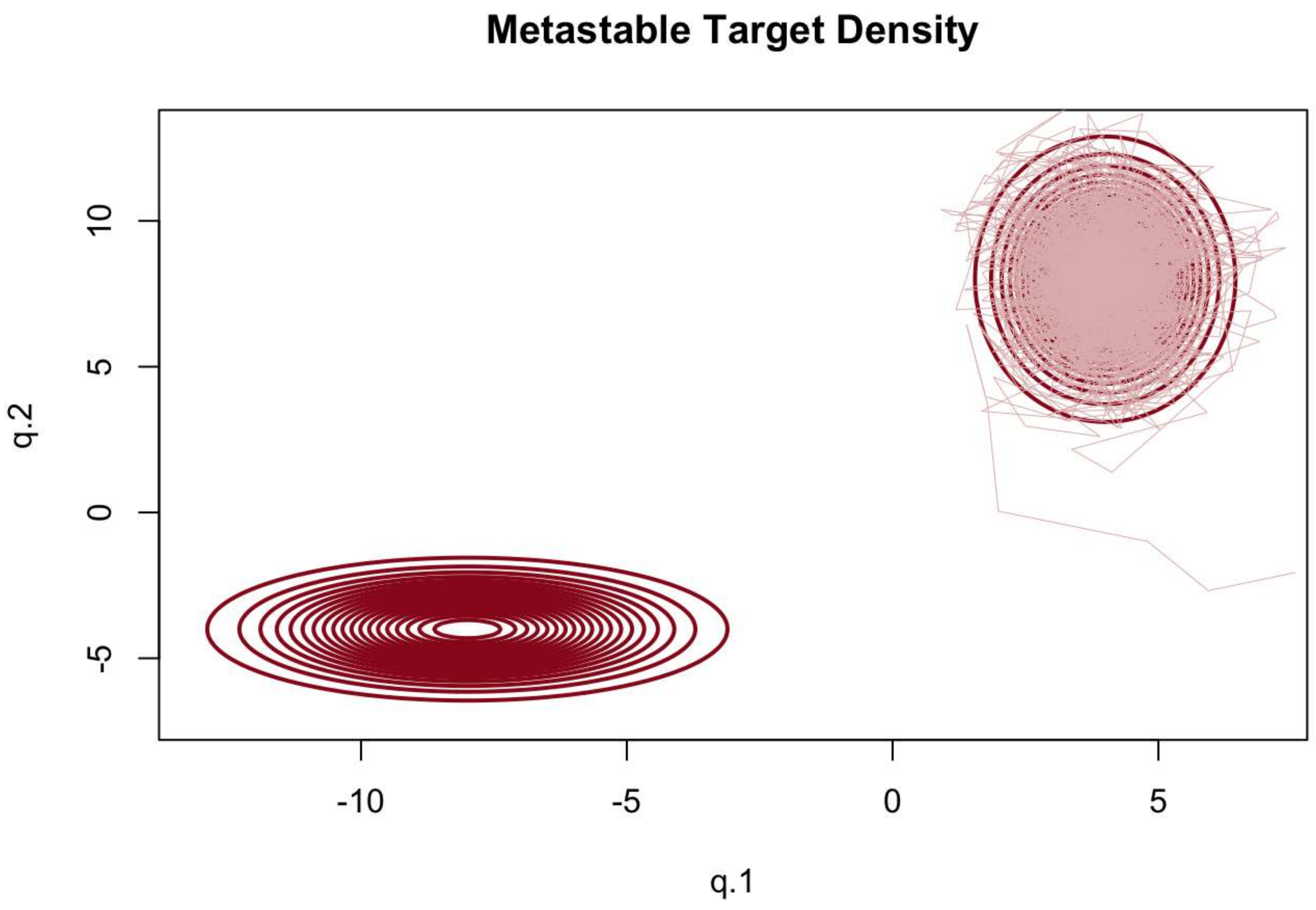
```
Model: gauss
chains=4, num_particles=1000, warmups=1000

     mean     sd     ess_bulk   r_hat
x   2.793   0.373        21.0    1.23
y   3.028   0.335        43.0    1.08
```



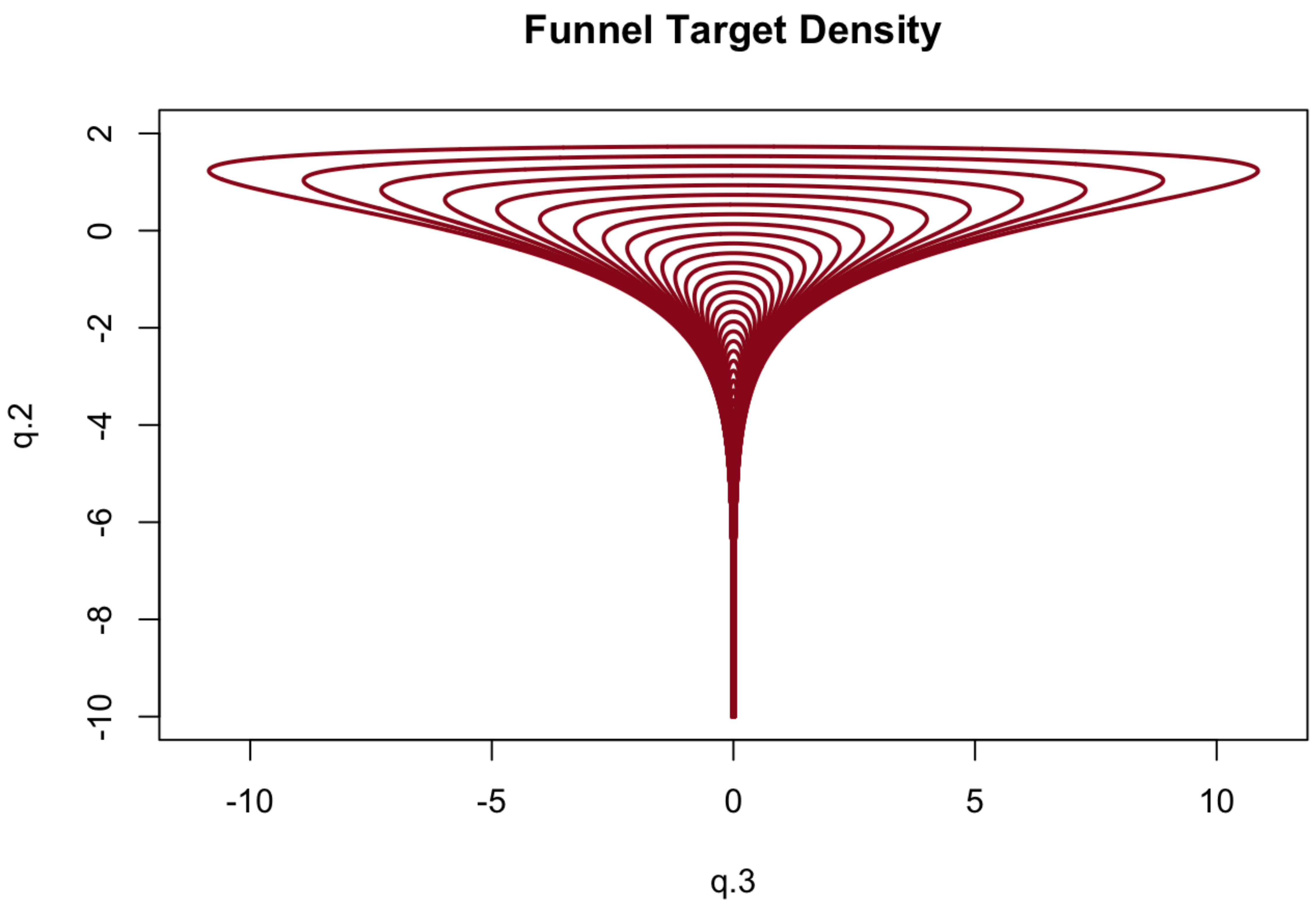


images from Vehtari et al. 2021

Pathological models



Multimodal distribution



Neal's funnel

images from Bettancourt 2020

Density semantics

Probabilistic Programming Languages

Reminders: deterministic semantics

functions

expression

value

$$[\![e]\!]^{\phi}_{\gamma} : V$$

environment

Example:

$$\begin{aligned}[\![\texttt{let}\ y\ \texttt{=}\ 40\ \texttt{in}\ x + y]\!]^{\emptyset}_{[x \leftarrow 2]} &= [\![x + y]\!]^{\emptyset}_{[x \leftarrow 2, y \leftarrow 40]} \\ &= 42\end{aligned}$$

Reminders: kernel semantics

Unnormalized measure

functions

Measure over return values

expression

$$\{\!\!\{ e \}\!\!\}_{\gamma}^{\phi} : \texttt{Meas}(V)$$

environment

Example:

$$\{\!\!\{ \texttt{sample}(\mathcal{N}(0,1)) \}\!\!\}_{\emptyset}^{\emptyset}(\mathbb{R}^{+}) = 0.5$$

$$[\![\texttt{infer}(e)]\!]_{\gamma}^{\phi} = \frac{\{\!\!\{ e \}\!\!\}_{\gamma}^{\phi}}{\{\!\!\{ e \}\!\!\}_{\gamma}^{\phi}([\![\mathit{typeOf}(e)]\!])}$$

Density semantics

Key idea

- A model is a function $f : R \to t \times \mathbb{R}^+$
- Associate a value $v(r)$ and a score $W(r)$ to parameters (random variables)
- Deterministic function *given an oracle* for the parameters

Interpretation close to our weighted samplers for approximate inference

Back to measure

- ρ: uniform distributions on parameters
- We get a measure by integrating f along ρ

$$\mu(U) = \int \rho(dr) W(r)\, \delta_{v(r)}(U)$$

Problem: random variables can change between two executions

```
let c = sample (bernoulli ~p:0.5)
if c then let x = sample (gaussian ~mu:0. ~sigma:1.) in ...
```

Measure over parameters

Key ideas

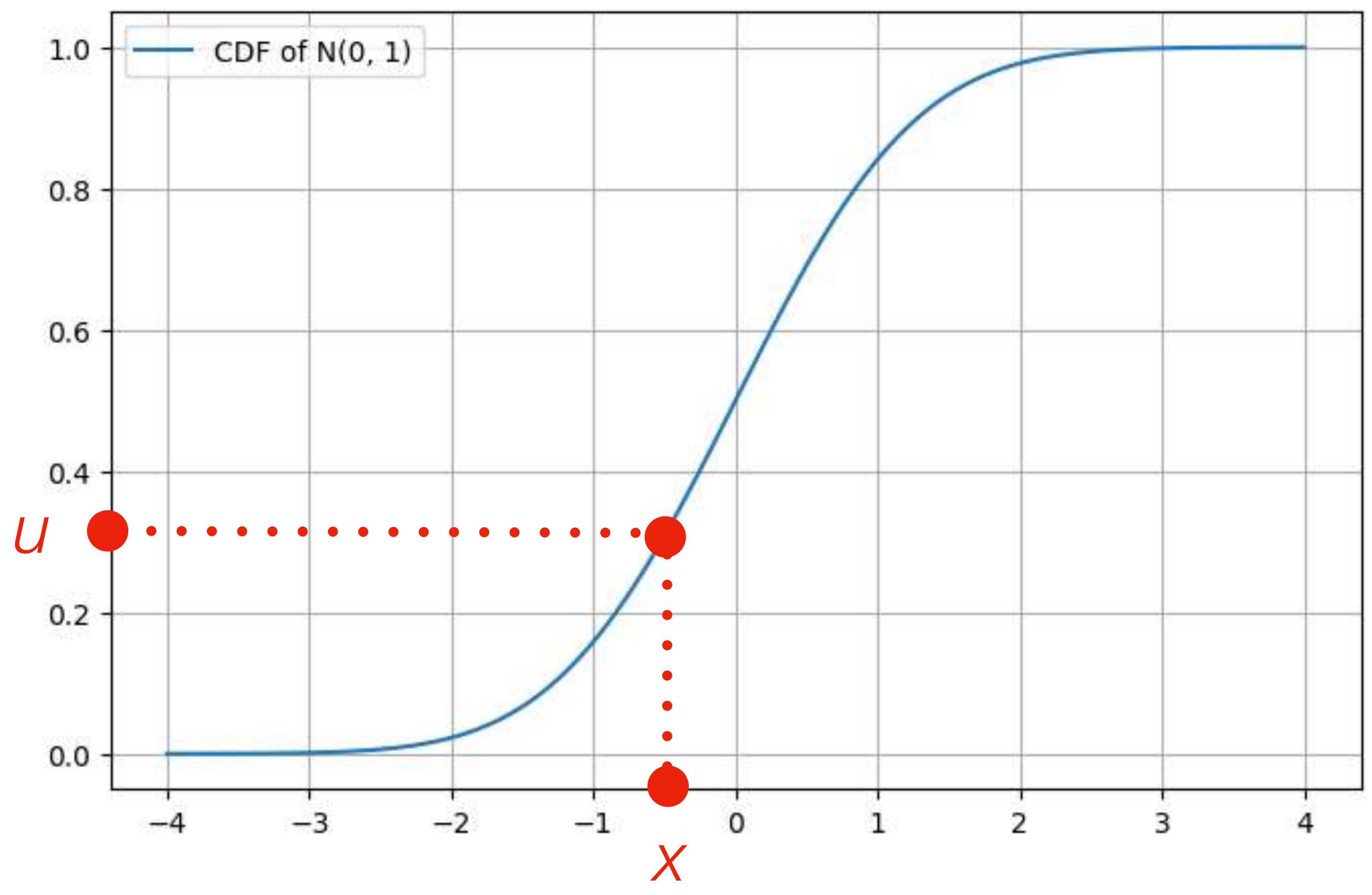
- Map random elements in [0, 1] to samples using inverse transform sampling
- Pass random elements as argument to the semantics

Inverse transform sampling

- Draw a sample $u \sim \textit{Uniform}(0, 1)$
- Compute sample value $x = \textit{icdf}(\mu)(u)$
- $\textit{icdf}(\mu)$: generalized inverse of the cumulative distribution function

Uniform measure over parameters

- Use the Lesbegue measure λ over $[0, 1]$ for each random element
- Product space across all possible parameters (cube)



Lee et al. 2020

Density semantics

functions

expression　　random elements　　score

$$(\!|e|\!)^{\phi}_{\gamma}(r) : V \times \mathbb{R}^{+}$$

return value

environment

Example:

$$(\!|\texttt{sample}(\mathcal{N}(0,1))|\!)^{\phi}_{\gamma}(r) = \textit{icdf}(\mathcal{N}(0,1))(r)$$

$$[\![\texttt{infer}(e)]\!]^{\phi}_{\gamma}(U) = \begin{cases} \dfrac{\mu(U)}{\mu(\top)} \quad \textit{with} \begin{cases} \mu(U) = \int \rho(dr)\, W(r)\, \delta_{v(r)}(U) \\ v(r), W(r) = (\!|e|\!)^{\phi}_{\gamma}(r) \end{cases} & \textit{if } 0 < \mu(\top) < \infty \\ \text{Error} \quad \textit{otherwise} \end{cases}$$

Density semantics

Remember: sampler semantics

- Expressions are interpreted as weighted samplers in log space
- Given an environment ϕ, γ, and random elements r, $(\!|e|\!)^{\phi}_{\gamma}(r) = v, W$
- $(\!|e|\!)^{\phi} : \Gamma \times R \to V \times \mathbb{R}^{+}$
- Parameters are now inputs

$$
\begin{aligned}
(\!|c|\!)^{\phi}_{\gamma}(\emptyset) &= c, 1 \\
(\!|x|\!)^{\phi}_{\gamma}(\emptyset) &= \gamma(x), 1 \\
(\!|\texttt{sample}(e)|\!)^{\phi}_{\gamma}(r) &= \mathit{icdf}([\![e]\!]^{\phi}_{\gamma})(r), 1 \\
(\!|\texttt{factor}(e)|\!)^{\phi}_{\gamma}(\emptyset) &= (), [\![e]\!]^{\phi}_{\gamma} \\
(\!|\texttt{observe}(e_1, e_2)|\!)^{\phi}_{\gamma}(\emptyset) &= (), \mathrm{pdf}([\![e_1]\!]^{\phi}_{\gamma})([\![e_2]\!]^{\phi}_{\gamma}) \\
(\!|\texttt{let}\ x = e_1\ \texttt{in}\ e_2|\!)^{\phi}_{\gamma}(r_1, r_2) &= \mathit{let}\ v_1, W_1 = (\!|e_1|\!)^{\phi}_{\gamma}\ \mathit{in}\ (r_1) \\
&\quad\ \mathit{let}\ v_2, W_2 = (\!|e_2|\!)^{\phi}_{\gamma+[x \leftarrow v_1]}\ \mathit{in}\ (r_2) \\
&\quad\ v_2, W_1 \times W_2
\end{aligned}
$$

Random elements: program structure

Deterministic expression: $\emptyset$

Sub-expressions: nested tuples

must know the structure of the program...

Exercises

What is the density semantics of the following programs?

```
let my_gaussian (mu, sigma) =
  let x = sample (gaussian (mu, sigma)) in
  x

let my_beta (a, b) =
  let x = sample (uniform (0., 1.)) in
  let () = observe (beta (a, b), x) in
  x

let coin (x1, ... , xn) =
  let z = sample (uniform (0., 1.)) in
  observe (bernoulli (z), x1); ... ; observe (bernoulli (z), xn);
  z
```

Semantics equivalence

Theorem. *For an expression* e*, the density semantics of* e *is the density of the measure defined by the kernel semantics.*

$$\{\!\![e]\!\!\}_{\gamma}^{\phi}(U) = \int \rho(dr) W(r) \delta_{v(r)}(U) \qquad \textit{where} \;\; v(r), W(r) = (\!| e |\!)_{\gamma}^{\phi}(r)$$

Proof. By induction... (see notes) $\square$

The kernel and density semantics define the same object

Advanced inference: HMC, SVI

Probabilistic Programming Languages

Hamiltonian Monte-Carlo (HMC)

Preferred inference algorithm for Stan

Analogy: Particle in an energy field

- Program define a density of the form exp$(-U(X))$
- On continuous spaces U can be interpreted as an energy
- Low energy wells correspond to high probability regions
- HMC simulate the trajectory of a particle in this energy field

Hamiltonian Dynamics

- M: mass matrix
- P: momentum

$$K(P) = \frac{1}{2} P^T M^{-1} P$$

$$\underbrace{H(X, P)}_{\text{hamiltonian}} = \underbrace{K(P)}_{\text{kinetic energy}} + \underbrace{U(X)}_{\text{potential energy}}$$

Hamiltonian Monte-Carlo (HMC)

Energy conservation

$$\frac{dH}{dt} = (\nabla_P H)^T \frac{dP}{dt} + (\nabla_X H)^T \frac{dX}{dt}$$

Hamiltonian dynamics

$$\begin{cases} \dfrac{dX}{dt} = \nabla_P H(X, P) = M^{-1} P \\ \dfrac{dP}{dt} = -\nabla_X H(X, P) = -\nabla_X U(X) \end{cases}$$

$$\overset{\text{hamiltonian}}{\uparrow}\!\!\!\!\!\!\!\! \overset{\text{hamiltonian}}{H}(X, P) = \underset{\text{kinetic energy}}{K}(P) + \overset{\text{potential energy}}{U}(X)$$

Hamiltonian Monte-Carlo (HMC)

Generate samples (X, P) from the density $\exp(-H(X, P))$

- At each iteration
- Sample an initial momentum $P_0 \sim \mathcal{N}(0, M)$
- Solve the Hamiltonian dynamics (discretized)
- Perform a Metropolis Hastings update with probability α

momentum can then be marginalized

$$\alpha = \min\left(1, \frac{\exp(-H(X_i, P_i))}{\exp(-H(X_{i-1}, P_{i-1}))}\right)$$

If the hamiltonian is preserved: accept with probability 1.

- Problem: numerical approximations
- Solution: leapfrog integrator and reject using MH acceptance probability

Hamiltonian Monte-Carlo (HMC)

```
let u x = let _, w = model data x in w
let k p =  0.5 * transpose p * inv m * p

let h x p = u x +. k p

let rec gen n values x =
  if n = 0 then values
  else
   let p = Distribution.draw mv_normal(0, m) in
   let x', p' = leapfrog (grad u) x p in
   let next_x = if Random.float 1. < exp(h x p -. h x' p') then x' else x in
   let next_value, _ = model data next_x in
   gen (n - 1) (next_value :: values) next_x
```

model is a function from parameters to (value, score), differentiable.

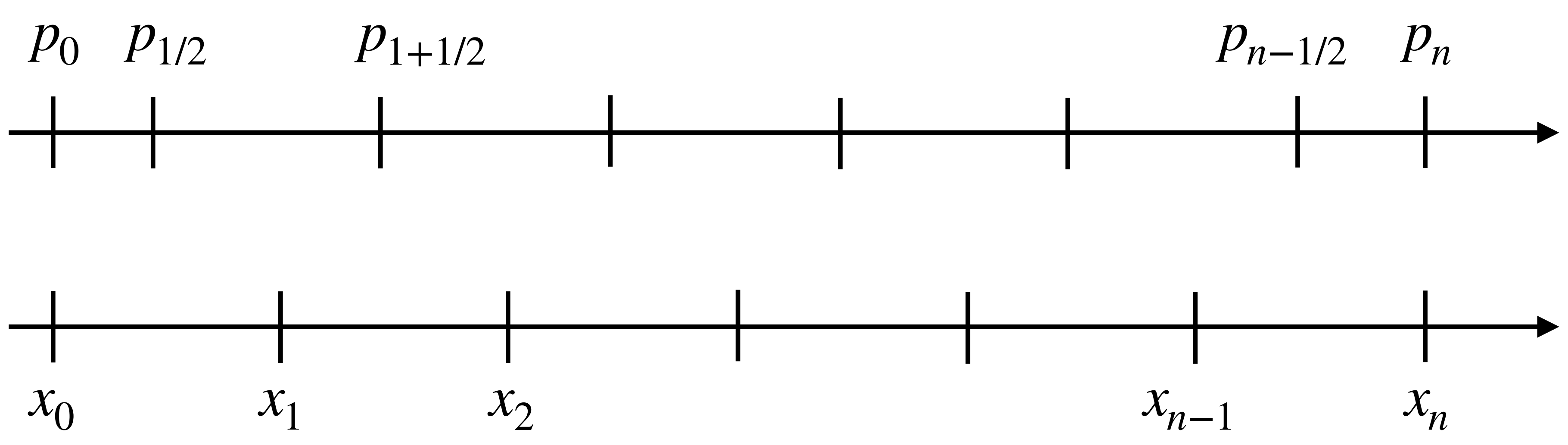
autodiff magic!

Warning: pseudo-code

Leapfrog integration

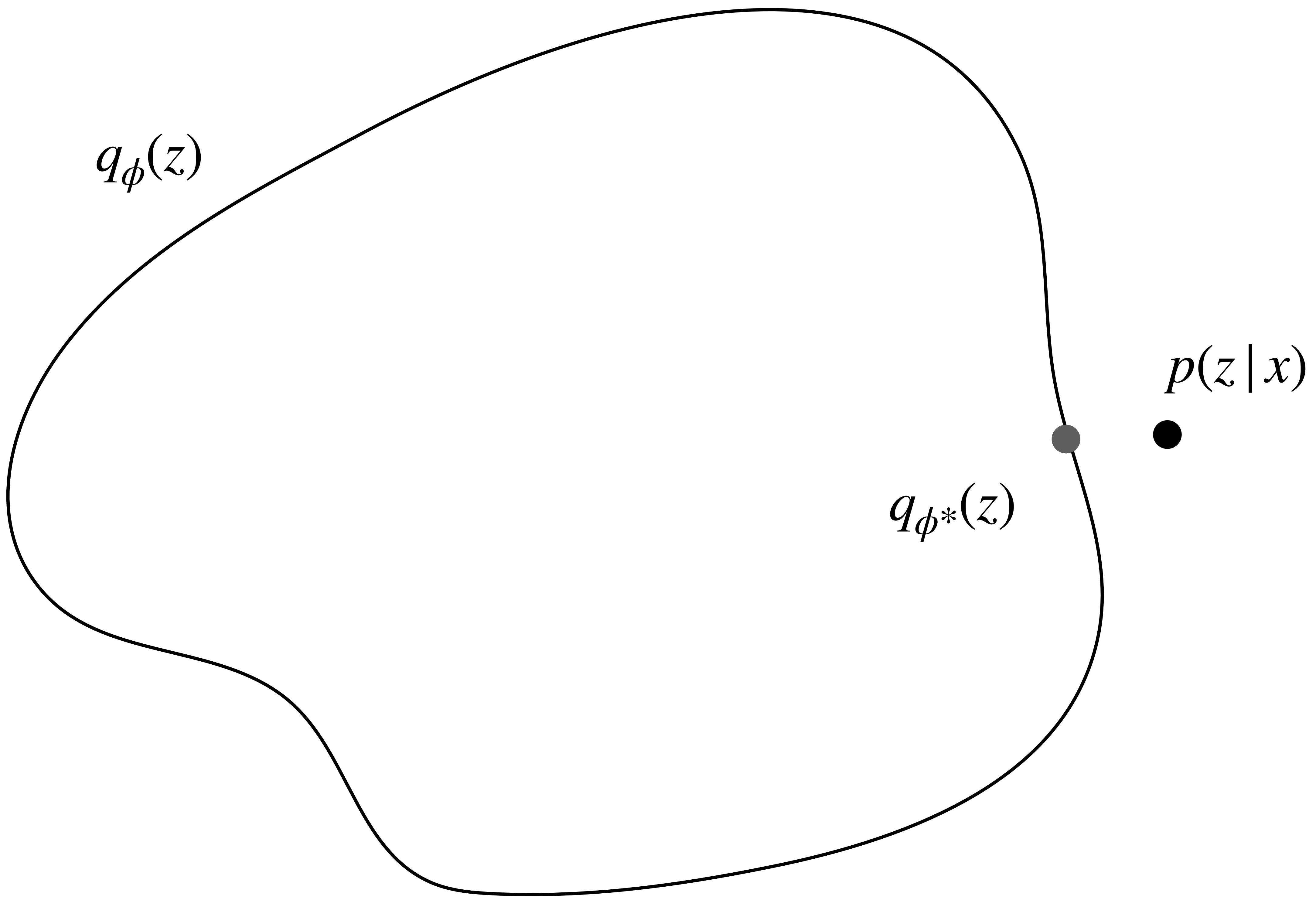
```
let leapfrog u_grad x0 p0 =
  let p = p0 - 0.5 * step_size * u_grad x0 in       (* first half step for the momentum *)
  let rec loop n x p =
    if n = 0 then x, p
    else
      let x' = x + step_size * p in
      let p' = p - step_size * u_grad x' in
      loop (n-1) x' p'
  in
  let xt, pt = loop (path_len - 1) x0 p in
  let x' = xt + step_size * pt in
  let p' = pt - 0.5 * step_size * u_grad x' in       (* last half step for the momentum *)
  x', p'
```

Warning: pseudo-code



Stochastic Variational Inference (SVI)

$$p(z \mid x) = \frac{p(x \mid z)p(z)}{p(x)} = \frac{p(x \mid z)p(z)}{\int_z p(x \mid z)p(z)dz}$$



Variational family

- Parameterized by a parameter ϕ
- Find the closest member to the posterior $q_{\phi^*}(z)$
- Optimization problem

Metrics: Kullback-Leibler divergence

$$KL(q(x) \parallel p(x)) = -\int q(x) \log \frac{p(x)}{q(x)}$$

- $KL(q \parallel p) \geq 0$ positive
- $KL(q \parallel p) = 0 \iff |x| \neq 0 \implies p(x) = q(x)$, equal almost everywhere
- $KL(q \parallel p) \neq KL(p \parallel q)$ asymetric
- No triangular inequality

Blei et al. 2017

Stochastic Variational Inference (SVI)

$$
\begin{aligned}
KL(q_\phi(z) \parallel p(z|x)) &= -\int q_\phi(z) \log \frac{p(z|x)}{q_\phi(z)}\ dz \\
&= -\int q_\phi(z) \log \frac{p(x,z)}{p(x)q_\phi(z)}\ dz \\
&= -\int q_\phi(z) \log \frac{p(x,z)}{q_\phi(z)}\ dz + \int q_\phi(z) \log p(x)\ dz \\
&= -\int q_\phi(z) \log \frac{p(x,z)}{q_\phi(z)}\ dz + \log p(x)
\end{aligned}
$$

Stochastic Variational Inference (SVI)

$$
\begin{aligned}
KL(q_\phi(z) \parallel p(z|x)) &= -\int q_\phi(z) \log \frac{p(z|x)}{q_\phi(z)} \, dz \\
&= -\int q_\phi(z) \log \frac{p(x,z)}{p(x) q_\phi(z)} \, dz \\
&= -\int q_\phi(z) \log \frac{p(x,z)}{q_\phi(z)} \, dz + \int q_\phi(z) \log p(x) \, dz \\
&= -\int q_\phi(z) \log \frac{p(x,z)}{q_\phi(z)} \, dz + \log p(x)
\end{aligned}
$$

$$
\underbrace{\log p(x)}_{\text{constant}} = \underbrace{KL(q_\phi(z) \parallel p(x \mid z))}_{\text{minimize}} + \underbrace{\int q_\phi(z) \, \log \frac{p(x,z)}{q_\phi(z)} \, dz}_{\text{maximize ELBO}}
$$

Stochastic Variational Inference (SVI)

How to solve the optimisation problem?

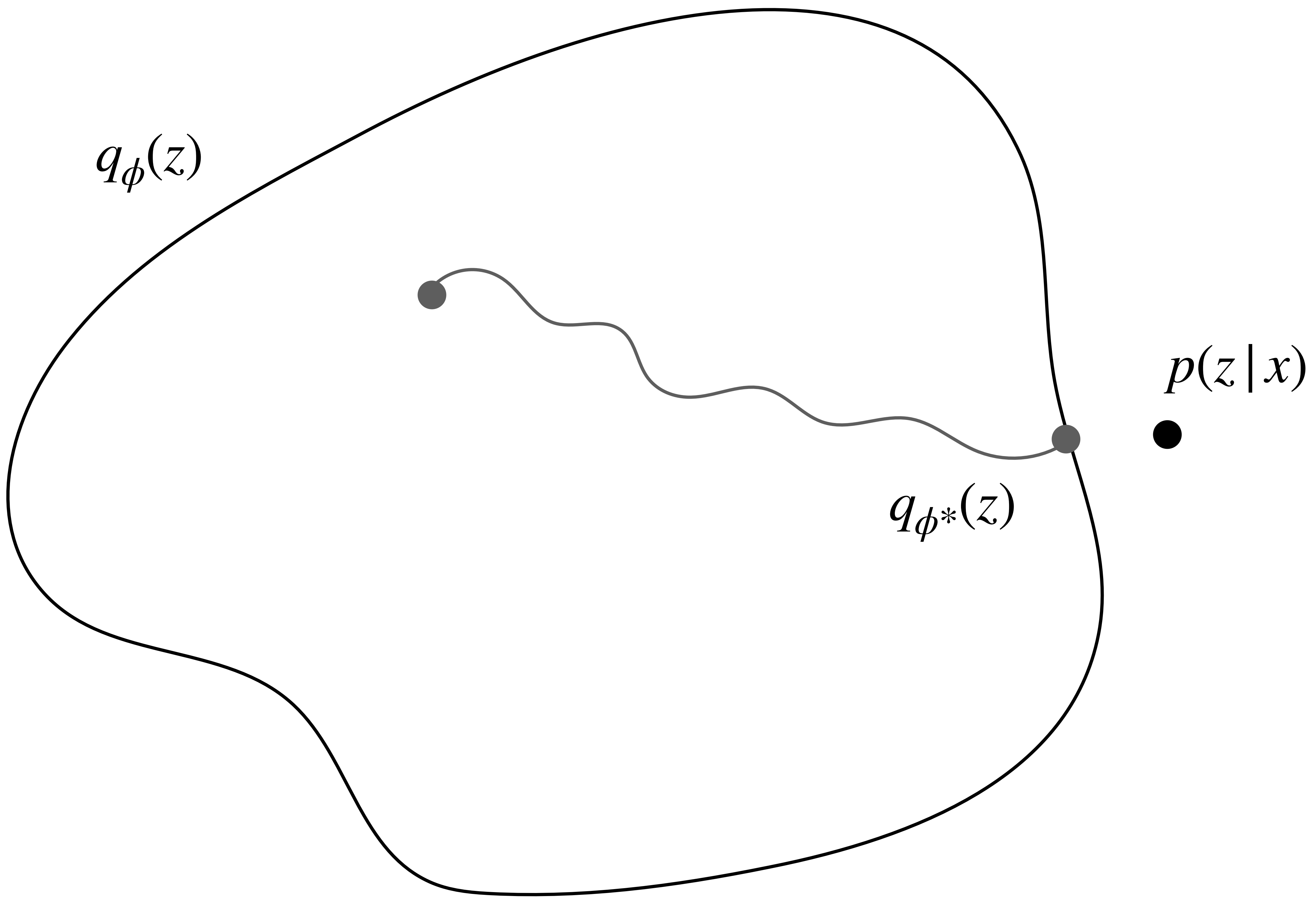
Program your own guide

- Pyro (first versions)
- Sample the same variables in the guide and the model

```
def model():
  pyro.sample("z_1", ... )

def guide():
  pyro.sample("z_1", ... )
```

Approximate gradient ascent.

$$\nabla_\phi KL(q_\phi(z) \parallel p(z \mid x)) \longrightarrow \nabla_\phi \mathcal{L} = \nabla_\phi \int q_\phi(z) \ \log \frac{p(x,z)}{q_\phi(z)} \ dz$$



autodiff magic!

Lee, Yu, Rival, Yang POPL 2020

Stochastic Variational Inference (SVI)

How to solve the optimisation problem?

Black-box variational inference

- Variational families with tractable solution
- Mean-field approximation $q_\phi(z) = \prod_{i=1}^{n} \mathcal{N}(z_i|\mu_i, \sigma_i)$ where $\phi = \{\mu_i, \sigma_i\}_{i\in[1,n]}$
- Full-rank approximation $q_\phi(z) = \mathcal{N}(z|\mu, \Sigma)$ where $\phi = (\mu, \Sigma)$

Assumptions

- Independences between random variables
- Only use Gaussians distributions

Blei et al. 2017

References

An Introduction to Probabilistic Programming
Jan-Willem van de Meent, Brooks Paige, Hongseok Yang, Frank Wood
https://arxiv.org/abs/1809.10756

Lightweight Implementations of Probabilistic Programming Languages Via Transformational Compilation
David Wingate Andreas Stuhlmüller Noah D. Goodman
AISTATS 2011

Markov Chain Monte Carlo in Practice
Michael Bettancourt
https://betanalpha.github.io/assets/case_studies/markov_chain_monte_carlo.html

Variational Inference: A Review for Statisticians
David M. Blei, Alp Kucukelbir, Jon D. McAuliffe
https://arxiv.org/abs/1601.00670